

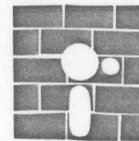
ILLUSTRATED BY CARY HENRIE

PROGRAMMING PROJECT

CALCULATING CRCs BY BITS AND BYTES

BY GREG MORSE

Use the XOR function to implement modulo 2 division when calculating cyclic redundancy checks



Recently I needed to implement the XMODEM cyclic redundancy check (CRC) option on my XCom9 modem program. Despite the many programs available for

calculating CRCs, I had some difficulty understanding the math behind the calculations. The details became clear after much research and experimentation.

The starting point for all CRCs is fancy linear algebra. The CRC is defined in terms of message polynomials, generator polynomials, and so on. As Perez, Wismer, and Becker state in "Byte-wise CRC Calculations" (*IEEE Micro*, June 1983),

In a system employing CRC the message being transmitted is considered to be a binary polynomial $M(X)$. It is first multiplied by X^k and then divided (modulo 2) by an arbitrary generator polynomial $G(X)$ of degree k which results in a quotient $Q(X)$ and a remainder. . .

It sounds confusing, but if you can understand how to apply the math, you can design more efficient programs and spot many erroneous ones.

THE "MATH-NESS" TO THE METHOD

The design of the polynomial $G(X)$ is extremely complex. You need to pick one that

produces CRCs that are good at detecting errors. Fortunately, many $G(X)$ s exist already. Table 1 contains the two most common $G(X)$ s in an 8-bit-byte environment.

Let's calculate the CRC for the letter T, 0101 0100 in binary. $M(X)$ is the message as it is transmitted. You transmit a character's least significant bit (LSB) first, so $M(X)$ becomes 00101010. Then you divide modulo 2 as shown in figure 1. (Modulo 2 means you use the XOR instruction instead of the normal add and subtract.) Work it through according to the process shown in figure 2. Note that the CRC result is given in reverse order, that is, most significant bit (MSB) on the right, LSB on the left.

BIT-ORIENTED ALGORITHMS

Using the long-division approach, if you had only a single zero bit to send, you would get the result shown in figure 3. If you had two bits to send, a zero and then a one, long division would produce the result shown in figure 4. The first remainder in figure 4 is the same as the first remainder in figure 3 except that its LSB has been XORed with

(continued)

Greg Morse (10871 Roseland Gate, Richmond, B.C., Canada V7A 2R1) is an engineer with the British Columbia Telephone Company. He has 20 years' experience with computers and 10 with data communications. Greg has B.A.Sc. and M.A.Sc. degrees in electrical engineering.

Table 1: The two most common $G(X)$ s (generator polynomials) in an 8-bit-byte environment. You can code the $G(X)$ as a 17-bit binary, or hexadecimal, number. The ones and zeros represent the coefficients of the different powers of X in the polynomial.

CRC-16 (Bisynchronous)
 $X^{16} + X^{15} + X^2 + 1 = 1\ 1000\ 0000\ 0000\ 0101$
 CRC-CCITT (SDLC, X.25, XMODEM)
 $X^{16} + X^{12} + X^5 + 1 = 1\ 0001\ 0000\ 0010\ 0001$

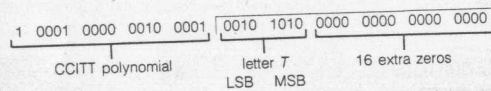


Figure 1: The initial modulo 2 division used to calculate the CRC for the letter T. Note that the letter is given LSB first; in other words, it is reversed.

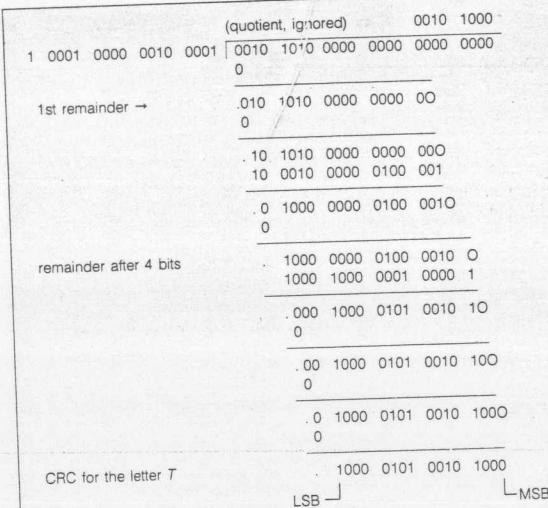


Figure 2: The entire long-division calculation process used to derive the CRC for the letter T. The apparent subtractions in this process are XORs; therefore, there are no "borrows." A "." represents a zero that has no further part in the calculation, and "O" represents a zero that was "brought down" from the dividend line. Therefore, the line .00 1000 0101 0010 100 represents a remainder of 00 1000 0101 0010 10. Note that the CRC is given with the LSB on the left. Thus, the CRC for the letter T = 0 54 is (MSB first) 0001 0100 1010 0001, or 14A1 hexadecimal.

the second data bit. This similarity will always be true because of the way the $M(X)$ polynomial is built.

This observation leads to the following bit-by-bit algorithm for calculating the CRC:

1. Write down the first data bit (zero or one) to be transmitted.
2. Write down 16 zeros to its right.
3. Divide the 17-bit number by the 17-bit CRC polynomial using XOR instead of subtraction. Make a note of the remainder, which is the CRC.
4. Get the next data bit.
5. XOR this bit with the LSB (left-most bit) of the CRC in step 3.
6. Append a zero to the right-hand end of the result in step 5.
7. Divide the 17-bit number from step 6 by the 17-bit CRC polynomial. Use XOR instead of subtraction. The remainder is the CRC.
8. Repeat steps 4 through 7 until there are no more data bits. (You can replace steps 1 through 3 with a single step to initialize the CRC to zeros.)

Thus, you can calculate the CRC a bit at a time, for any number of bits, with almost no extra calculation overhead. In summary, the steps involved in the long-division method are

1. The message bits are written down in the order in which they are transmitted, from left to right, that is, LSB on the left.
2. Sixteen zeros are appended to the right-hand end of the binary number formed in step 1.
3. The generating polynomial is written down MSB first, that is, on the left.
4. The division is done modulo 2, that is, using XOR instead of normal subtraction.
5. The CRC is the remainder after all data bits have been processed. The LSB is on the left.

Figure 5 translates these steps into a flowchart.

HARDWARE IMPLEMENTATIONS

One disadvantage of the flowchart in figure 5 is that it requires a 17-bit reg-

(continued)

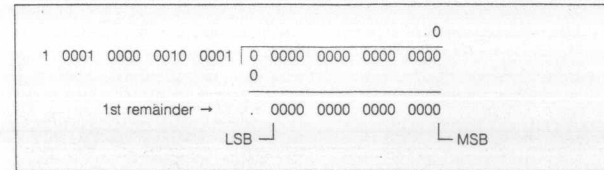


Figure 3: The long-division CRC calculation for a single zero bit.

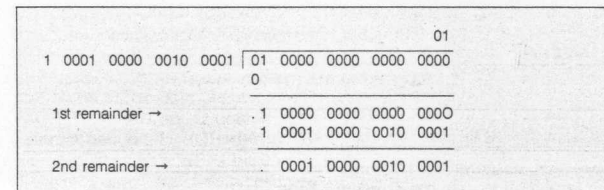


Figure 4: The long-division CRC calculation for the two bits 01.

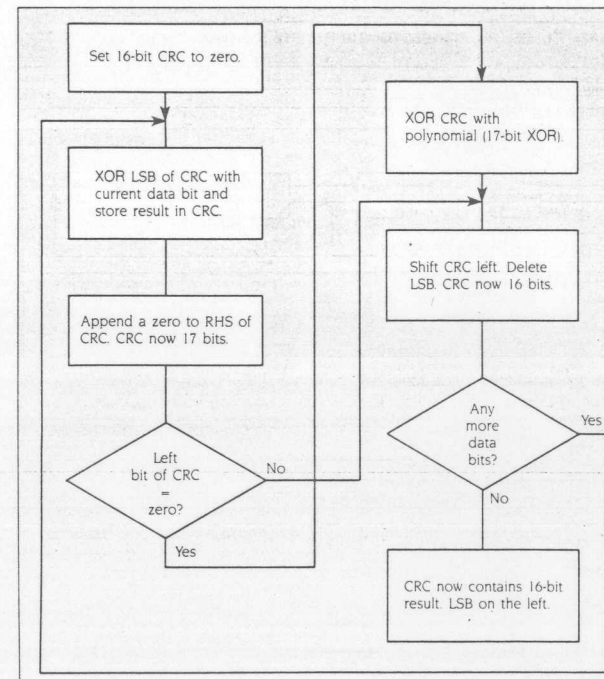


Figure 5: A flowchart representing the steps involved in the long-division method of calculating a CRC.

MathCAD

The Engineer's Scratch Pad

FOURIER RECONSTRUCTION: SQUARE WAVE

$N := 40$ $J := 0..N$... 40 points

$S_j := 1 - 2 \cdot \frac{j}{N} \cdot 20$... step function

Reconstruct function, using first two terms $i := 1, 3 \dots$

of Fourier series

$R_j := \frac{4}{N} \sum_{i=1}^2 \frac{1}{i} \sin \left[i \cdot 2 \cdot \pi \cdot \frac{j}{N} \right]$

R_j vs J

1.5
1.0
0.5
0
-0.5
-1.0
-1.5

0 J N

A powerful computation and documentation tool for your IBM-PC.

With MathCAD you simply and interactively create, edit and display formulas on the screen the way you are used to writing them. Equations are instantly computed and the results displayed on the screen as a single number or a plot. Text may be added to the screen and everything may be printed out as an integrated document. MathCAD has built-in hyperbolic and circular functions, performs all calculations with real and complex numbers, performs iterative calculations, handles all units, performs error checking and dimensional analysis and much more...

The price of MathCAD—\$189. In Massachusetts add 5% sales tax.

To order send check, p.o., call us with your MasterCard number or call us for the nearest dealer.

1.800.MathCAD or 617.577.1017

Math Soft

Σ + √ - = × ∫ ÷ δ

One Kendall Square
 Cambridge, Massachusetts 02139

Q=1	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	0
XOR																
X16=1	0	0	0	X12	0	0	0	0	0	0	X5	0	0	0	0	1
New CRC	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15

Figure 6: The division process if Q = 1.

Q=0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	0
XOR																
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
New CRC	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15

Figure 7: The division process if Q = 0.

Q=(D XOR R0)	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15	0
XOR																
X16	0	0	0	Q	0	0	0	0	0	0	Q	0	0	0	0	Q
New CRC	R0	R1	R2	R3	R4	R5	R6	R7	R8	R9	R10	R11	R12	R13	R14	R15

Figure 8: The division process regardless of the value of Q.

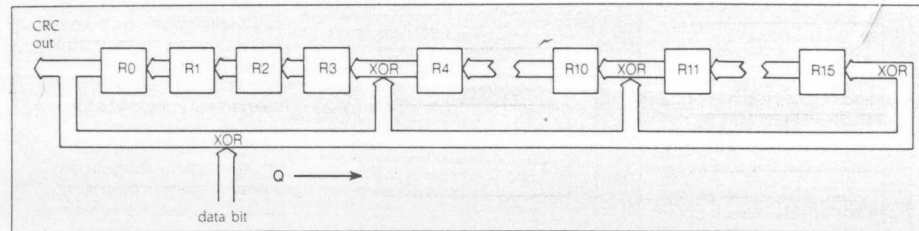


Figure 9: The process shown in figure 8 translated into a hardware circuit using shift registers and XOR gates. (The boxes are stages to a shift register; the shift register is only 16 stages long.) Note that the XOR gate going into R15 is superfluous, since Q XOR 0 is always Q. Note also that in the long-division method, the calculations are done before "bringing down" the next zero bit. Similarly here, the XORs are performed before the shift.

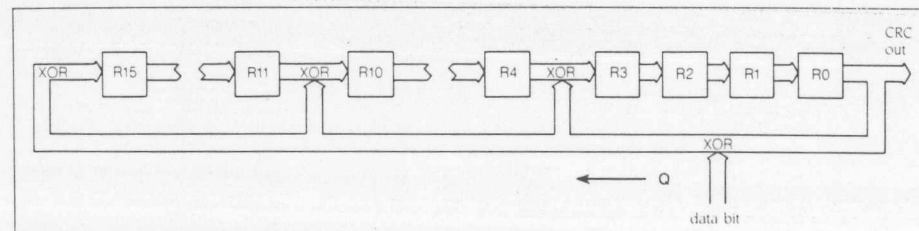


Figure 10: The reverse drawing of figure 9; the LSB is on the right instead of on the left.

ister. However, after the division is complete, the left bit will always be a zero. Either it was a zero to begin with, or if it was a one, it was XORed with the X^{16} bit of the polynomial, which is also a one, producing a zero. Thus, you can XOR with only the 16 LSBs of the polynomial. In the case of the CRC-CCITT polynomial, $X^{16} + X^{12} + X^5 + 1$ (1 0001 0000 0010 0001 in binary or 1 1021 in hexadecimal), you can XOR with 0 1021.

Let the bits of the CRC register be R0, R1, up to R15; let the data bits be D0 up to D7; let the polynomial bits be X0 up to X15; and define $Q = D \text{ XOR } R0$, that is, set Q equal to the current data bit, D, XORed with R0. In every case, the LSB is bit 0.

Then, if $Q = 1$, the division process looks like that in figure 6. If $Q = 0$, XOR R0 is zero, then the division reduces to that in figure 7. But when $Q = 1$, the new R15 is always 1, and you XOR R4 and R11 with X12 and X5, which are 1. If $Q = 0$, then the new R15 is zero and you XOR R4 and R11 (and all other Rns) with zero.

You can combine the two cases as shown in figure 8. Turning that process into a circuit using shift registers and XOR gates (see figure 9) is straightforward. You can also draw the circuit so that the LSB (R0) is on the right (see figure 10). Both diagrams are convenient depending on the kind of software algorithm to be derived. Note that in hardware the XOR is done before the shift because of the way flip-flops work. The important point is that the new R10 = (old R11) XOR (old R0) XOR (new data bit).

BIT-BY-BIT SOFTWARE ALGORITHMS

When you do the calculations in software, you don't have to do the XOR before the shift. The important thing in software is to avoid having to deal with 17-bit values that don't fit into a variable. A software routine also doesn't need to use the $Q = D \text{ XOR } R0$ result to drive a gate. You can program the polynomial directly into the code as a constant.

You can now derive a software routine based on figure 10 (or figure 9, in which the order of storing the bits of the CRC is reversed). If you assume

that the CRC result is kept in a 16-bit integer with R15 (the MSB of the CRC) in the high (leftmost) bit position, then if $Q = 1$, you first shift the previous CRC to the right as in figure 11. In terms of a high-level language, you shift the CRC right by one, discarding the LSB, and XOR with 0 8408. By

testing Q first and then doing the shift before the XOR, you avoid the need for a 17-bit register. Because of the way in which the CRC is stored in the variable, the XOR is done with 0 8408 rather than 0 1021, as you might expect. This follows directly from the

(continued)

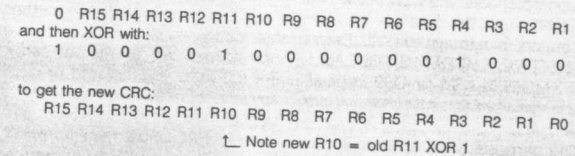


Figure 11: The diagram for a software routine based on the hardware circuit in figure 10. When $Q = 1$ the new CRC becomes the previous CRC shifted right by one bit position and XORed with G(X).

Listing 1: CCITT, the CCITT routine for calculating CRCs in C source code.

```

/* Straightforward, non-optimized CRC-CCITT routine */
/* Assumes 16-bit integer variables */
/* MSB of integer is MSB of CRC result */
#define POLY 0x8408
/* POLY = 1021 in bit rev order */
BLKCRD(bufptr, crcres, count)
unsigned char *bufptr;
unsigned int *crcres, count;
{
    int i;
    *crcres = 0;
    for (i=1; i<=count; ++i, bufptr++) /* for SDLC use 0xFFFF */
        bytcrd(bufptr, crcres) /* do for whole BLK */
        return (*crcres); /* do CRC for 1 char */
} /* end BLKCRD */

bytcrd(bufptr, crcres)
unsigned char *bufptr;
unsigned int *crcres;
{
    unsigned int j,ch,Q;
    ch = (unsigned int) *bufptr; /* get char, to int fmt */
    for (j=1; j<=8; j++) /* do each bit LSB 1st */
        Q = (*crcres & 0x0001) ^ (ch & 0x0001) /* Q=R0 XOR D */
        if (Q == 0x0001) /* Q is one */
            *crcres = *crcres >> 1; /* shift right one */
            *crcres = *crcres ^ POLY; /* XOR with number */
        else /* Q is zero */
            *crcres = *crcres >> 1; /* just shift no XOR */
            ch = ch >> 1; /* move next data bit */
    } /* end FOR - data bits all done */
    return (*crcres);
} /* end bytcrd */

```

diagram shown in figure 10. Generally speaking, if you process data LSB first, you can store the CRC with its MSB in the MSB position of the integer variable.

Listing 1 contains the implementation of a CRC calculation based on the CCITT polynomial. CCITT provides lots of intermediate results, for clarity rather than performance. This program implements the CCITT/IBM FCS calculation in a standard way with one exception: initializing the CRC. [Editor's note: CCITT, XMODEM.C, SDLC.ASM, and XMODEM.ASM for OS9 are available in several formats; see the insert card after page 368.]

The purpose, however, is to implement the de facto XMODEM standard. I found no published XMODEM CRC specification as such. What has been published is a C program,

XMODEM.C, that does the calculations (see listing 2). Apart from using the CCITT polynomial, this program is not CCITT standard in that data is processed MSB first, rather than LSB first, and the CRC is initialized to zeros rather than to ones.

CHOICES IN CRC CALCULATION AND TRANSMISSION

A CRC calculated by these methods has some very desirable error-detection properties, but it is not perfect. For example, you can add or delete any number of zero bits to or from the beginning of the block without affecting the CRC. (The remainder stays zero no matter how many zero bits start the block.) Furthermore, since the CRC is a cyclic code, any error, such as a clock slippage, that deletes

a bit at the beginning of the block and inserts the same bit at the end of the block (the last bit of the CRC) will not affect the CRC. For these reasons the CRCs used by IBM in the SDLC protocol and CCITT in the X.25 and HDLC protocols specify the following:

1. All bits of a block are protected by the CRC.
2. The data is sent LSB first. The CRC is calculated on bits as they are sent.
3. The CRC is initialized to all ones. This allows detection of any missed or inserted zero bits at the beginning of a block. (Missed or inserted ones are still detected.)
4. The one's complement of the

(continued)

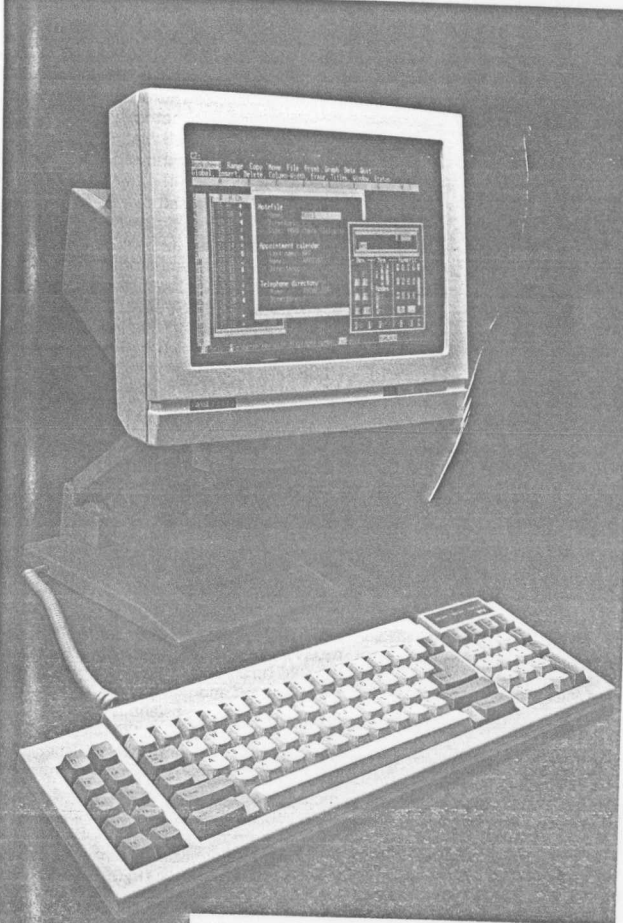
Listing 2: XMODEM.C, the XMODEM routine for calculating CRCs in C source code.

```
/*      Sample bit-oriented CRC routine      */
/*      Adopted from XMODEM protocol reference */

/* Calculate CRC on a block of data.          */
/* Ptr points to block of characters, count gives size of buffer. */
/* This program returns the CRC with the LSB of the CRC in the */
/* high bit of the result integer.             */
/* XMODEM deviates from the CCITT standard in that it does not use */
/* the LSB of the data 1st, nor does it initialize the CRC to all */
/* ones as specified by the standard.          */

int calcrc(ptr, count)
char *ptr;
int count;
{
    unsigned int crc;
    int i;
    crc = 0; /* note not 0xFFFF */
    while (--count >= 0) {
        i = (int) *ptr++; /* convert data char to int */
        i = i << 8; /* shift char to high byte */
        crc = crc ^ i; /* add current data to current */
        /* remainder modifies only least */
        /* sig 8 bits (high byte) of CRC */
        for (i=0; i<8; ++i) {
            if (crc & 0x8000) {
                crc = (crc << 1); /* loop for each bit */
                /* test D XOR R0 */
                /* discard LSB of CRC and */
                /* append zero */
                crc = crc ^ 0x1021; /* XOR with low 16 bits */
                /* of CCITT polynomial */
                /* because CRC is stored LSB 1st */
                /* polynomial written MSB 1st */
            }
            /* endif */
        }
        else
            crc = crc << 1; /* discard LSB & append 0 */
    }
    /* end while */
    return (crc & 0xFFFF); /* 16-bit result for whole block */
}
/* end calcrc */
```

The single best way to turn your PC-AT into a multi-user system.



Introducing the Wyse WY-60

Now there's a perfectly compatible, reliable, economical, Wyse way to get multi-user mileage from your PC-AT. Wyse WY-60 terminals give you complete compatibility for your IBM Personal Computer AT systems, right down to the exact keyboard layout, character set and display features.

The only thing different is how much cleaner and more readable your information is with the WY-60's high resolution and flat, non-glare, 14" tilt/swivel screen.

Multiple display formats go up to 132 columns and 44 lines on one screen, to get the most out of applications such as Multiplan and WordStar.

And a 512-character downloadable soft font is also there when you need mathematical symbols or customized character sets.

The adjustable arm is optional, and you can choose a green, white or amber screen.

No wonder we ship more terminals than anybody but IBM.*

Call toll-free or write, today, for more information. Wyse Technology, Attn: Marcom Department 60-AT, 3571 N. First St., San Jose, California 95134.

Call 1-800-GET-WYSE

WYSE

YOU NEVER REGRET A WYSE DECISION.



Wyse is a registered trademark of Wyse Technology. WY-60 and the "V" shaped design are trademarks of Wyse Technology. IBM and IBM Personal Computer AT are trademarks of International Business Machines Corporation. WordStar is a registered trademark of MicroPro International. Multiplan is a registered trademark of Microsoft Corporation. © 1986 Wyse Technology. *Equipment 1985 terminal shipment update.

Inquiry 383

Bit	*15	14	13	12	11	*10	9	8	7	6	5	4	*3	2	1	0
	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0

Figure 12: The contents of the shift register at the beginning of bitwise SDLC calculations.

Bit	*15	14	13	12	11	*10	9	8	7	6	5	4	*3	2	1	0
Shift 1	D0	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1
	R0					D0							D0			

Figure 13: The contents of the shift register after the first shift. Note that all entries in a column are XORed together.

Bit	*15	14	13	12	11	*10	9	8	7	6	5	4	*3	2	1	0
Shift 1	T0	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1
						T0							T0			

Figure 14: The same as figure 13 but with the abbreviation T0 for D0 XOR R0.

Bit	*15	14	13	12	11	*10	9	8	7	6	5	4	*3	2	1	0
Shift 4	T3	T2	T1	T0	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4
					T3	T2	T1	T0					T3	T2	T1	T0

Figure 15: The contents of the shift register after three more right shifts. T0 represents D0 XOR R0, T1 represents D1 XOR R1, and so on.

Bit	*15	14	13	12	11	*10	9	8	7	6	5	4	*3	2	1	0
Shift 5	T4	T3	T2	T1	T0	R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5
	T0					T4	T3	T2	T1	T0			T4	T3	T2	T1

Figure 16: The contents of the shift register after the fifth shift.

Bit	*15	14	13	12	11	*10	9	8	7	6	5	4	*3	2	1	0
Shift 8	T7	T6	T5	T4	T3	T2	T1	T0	R15	R14	R13	R12	R11	R10	R9	R8
	T3	T2	T1	T0		T7	T6	T5	T4	T3	T2	T1	T0	T7	T6	T5

Figure 17: The contents of the shift register after the eighth shift.

CRC is transmitted rather than the CRC itself. This allows detection of slippage-type errors.

5. The CRC is sent LSB first.

6. The polynomial used is $X^{16} + X^{12} + X^5 + 1$.

XMODEM.C implements the CRC as follows:

1. Only data bits are included. The header character and two block-number bytes are not included.
2. The data is sent LSB first. The CRC is calculated on the data MSB first.
3. The CRC is initialized to zeros.
4. The CRC is not complemented before transmission.
5. The order of transmission is high byte of CRC then low byte. Since UARTs transmit LSB first, this is equivalent to LSB of high byte through MSB of high byte, then LSB of low byte through MSB of low byte.
6. The polynomial used is $X^{16} + X^{12} + X^5 + 1$.

The CCITT method is easily implemented in a chip using shift registers and XOR gates, while the XMODEM method is not easily realized in hardware. To check an incoming data block, you have two options. You can calculate the CRC on all the protected bits only, omitting the CRC bits, and compare the calculated value to the received value. Or you can calculate the CRC on all the protected bits and the CRC itself and then compare the result to a known constant. If the first method is adopted in the CCITT case, the one's complement of the calculated value must be compared to the received CRC. In the XMODEM case, the comparison is direct. If the second method is adopted, in the XMODEM case the known constant is zero, while in the CCITT case it is 0 F0B8. (The high bit is on the left, i.e., 1.) No one's-complementing is required.

BYTE-ORIENTED SOFTWARE IMPLEMENTATIONS

The next step is to derive routines to calculate the CRC a whole byte at a time rather than bit by bit. This approach was first proposed by Perez, Wismer, and Becker. The motivation is that an 8-bit microprocessor is not

limited to single-bit XORs but can do them 8 bits at a time. The basic approach is to work from the hardware diagram (figure 9 or 10) and see what the CRC register would look like after 8 bits have been calculated.

Bitwise SDLC calculations. In figure 10 you can see that you have the contents of figure 12 in the shift register at the beginning of calculations. Then you take the LSB of data, D0, and XOR it with R0 (D0 XOR R0). After the right shift, the new R15 is D0 XOR R0, the new R10 is R11 XOR D0 XOR R0, and the new R3 is R4 XOR D0 XOR R0. (See figure 13).

The combinations D0 XOR R0, D1 XOR R1, D2 XOR R2, and so on, occur frequently, so let's abbreviate them as T0 = D0 XOR R0, T1 = D1 XOR R1, and so on. Figure 13 can now be rewritten as in figure 14. If you proceed in this fashion for three more shifts, you get figure 15.

To do the XOR on D4, you must use the content of the LSB of the shift register, which is now R4 XOR D0 XOR R0. The result is D4 XOR R4 XOR D0 XOR R0, or in shorter form, T4 XOR T0. The result after the fifth shift is shown in figure 16; after the eighth shift, in figure 17.

The tedious part is done. Now you want to write a program that will produce the same result when you're working with a byte of data as you would get from the shift register after eight shifts. The emphasis in this routine will be on speed. If you make it too general-purpose, a different choice of polynomial will lead to a completely different program. Also, for speed, it makes sense to code the routine in assembly language.

When working with 8-bit microcomputers, it is convenient to define the 8-bit quantities found in table 2. If you study figure 17, you will see that the combination

$$T7 \ T6 \ T5 \ T4 \\ \text{XOR} \ T3 \ T2 \ T1 \ T0$$

occurs several times. Let's call this term U = U7 U6 U5 U4. If you rewrite figure 17 with these substitutions, you get figure 18.

Further optimizations become apparent as you write the code. It is convenient to implement the SDLC algo-

Bit	*15	14	13	12	11	*10	9	8	7	6	5	4	*3	2	1	0
Line #																
1										R15	R14	R13	R12	R11	R10	R9
2	U7	U6	U5	U4	T3	T2	T1	T0								
3						U7	U6	U5	U4	T3	T2	T1	T0			
4														U7	U6	U5

Figure 18: The same as figure 17 but using the abbreviations given in table 2. Line 1 is CRCHi moved into CRCLo; line 2 is the high nybble of U and the low nybble of T; line 3 is the line 2 byte shifted left by 3 bits; and line 4 is U shifted right by 4 bits.

Table 2: Some convenient abbreviations for various 8-bit quantities used in CRC calculations in an 8-bit microcomputer environment.

CRCHi =	R15	R14	R13	R12	R11	R10	R9	R8
CRCLo =	R7	R6	R5	R4	R3	R2	R1	R0
Data =	D7	D6	D5	D4	D3	D2	D1	D0
T =	T7	T6	T5	T4	T3	T2	T1	T0
U =	U7	U6	U5	U4	0	0	0	0

Table 3: Some test cases provided for comparison if you want to write your own routine.

Text	SDLC	CRC
T	1B26	14A1
THE	44BE	7D8D
THE,QUICK,BROWN,FOX,0123456789	DF91	7DC5

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0

Figure 19: The contents of the shift register at the beginning of bitwise XMODEM calculations.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R15	R14	R13	R12	R11	R10	R9	R8	R7	R6	R5	R4	R3	R2	R1	R0
D7	D6	D5	D4	D3	D2	D1	D0								

Figure 20: The contents of the shift register after the XOR with the data byte.

rithm (as in the program SDLC.ASM) because of the built-in check that calculating a CRC on a "received" block provides; that is, the result should always be 0 F0B8.

The innermost loop of SDLC.ASM

takes only 86 cycles per byte including seven overhead cycles to check for end of buffer. If a 2-MHz 6809 were dedicated to CRC calculations, this would result in a through-

(continued)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
T7	T6	T5	T4	T3	T2	T1	T0	R7	R6	R5	R4	R3	R2	R1	R0

Figure 21: The same as figure 20 but with the abbreviations T7 for R15 XOR D7, T6 for R14 XOR D6, and so on.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
T6	T5	T4	T3	T2	T1	T0	R7	R6	R5	R4	R3	R2	R1	R0	T7

Figure 22: The contents of the shift register after processing the first data bit and performing the left shift.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
T3	T2	T1	T0	R7	R6	R5	R4	R3	R2	R1	R0	T7	T6	T5	T4

Figure 23: The contents of the shift register after four shifts.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
T2	T1	T0	R7	R6	R5	R4	R3	R2	R1	R0	T7	T6	T5	T4	T3

Figure 24: The contents of the shift register after five shifts.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R7	R6	R5	R4	R3	R2	R1	R0	T7	T6	T5	T4	T3	T2	T1	T0

Figure 25: The contents of the shift register after eight shifts.

Table 4: The same test cases as in table 3 but for XMODEM.

Text	XMODEM CRC
T	1A71
THE	1E0A
THE,QUICK,BROWN,FOX,0123456789	0498

put of about 200,000 bps. A bitwise algorithm to do the same job takes on average 353 cycles per byte and could take as many as 385. So the bitwise algorithm is about four times faster.

The test cases in table 3 are provided in case you want to write your own routine. The column labeled SDLC results when you initialize the CRC with ones (this is the result before the one's-complementing), and the column labeled CRC results when you initialize with zeros.

XMODEM byte-oriented algorithms. The XMODEM CRC "spec" such as it is, is shown in listing 2. If you repeat the work done for SDLC calculations (in figures 12 through 18) using listing 2 as a base, you will obtain the contents of the diagrams in figure 19. After the XOR with the data byte, you get the result shown in figure 20. When you abbreviate R15 XOR D7 as T7, R14 XOR D6 as T6, and so on, figure 20 becomes figure 21. After processing the first data bit and the left shift, you get the result shown in figure 22; after four shifts, figure 23; after five shifts, figure 24; and after eight shifts (the final result), figure 25. Next, you define the appropriate variables such as CRCHI, CRCLO, T, and U, and you look for factors.

An implementation of the XMODEM CRC calculations for the 6809 is given in the program XMODEM.ASM. The test case results using the XMODEM algorithm are shown in table 4.

CONCLUSION

Once you arrive at the basic idea of using the XOR function to implement modulo 2 division, it is not difficult to apply the CRC algorithms, only tedious. As with so many other communications standards, the options and the details provide most of the difficulties and confusion. ■

BIBLIOGRAPHY

Forsberg, Chuck, ed. "XMODEM/YMODEM Protocol Reference." Godzilla BBS, (503) 621-3746.
IBM SDLC General Information Manual, Appendix B.
McNamara, J. E. *Technical Aspects of Data Communications*. Bedford, MA: Digital Equipment Press, 1982.

SYSTAT

LESS BULK MORE STATISTICS!

How has SYSTAT become one of the largest statistical software companies in only 3 years?

LESS BULK:

- Will run from floppy disks
- Needs less memory (256K on IBM or compatible machines, 64K on C/P/M, 400K on VAX, or 512K on Apple Macintosh)
- Has fewer than 1/3 of the commands of other manufacturers' packages

MORE STATISTICS:

- Full screen spreadsheet data editor
- Missing data, arrays, character variables
- Unlimited cases
- Process rectangular, hierarchical, triangular files and variable length records
- Relational database management and file concatenation
- Character, numeric, and nested sorts
- Unlimited numeric and character transformations
- Interactive or batch
- Read and write text and external files
- Subgroup processing in statistical modules with SELECT and BY
- Value labels
- RECODE statements for quick multiple codes
- Scatterplots, contours, histograms, stem-and-leaf, boxplots, bar charts, quantile, probability plots
- Basic statistics, frequencies, T-tests
- Multi-way crosstabs with log-linear modeling, association coefficients, PRE statistics, asymptotic standard errors
- Pairwise/listwise missing value correlation, SSCP, covariance, Spearman, Gamma, Kendall Tau, Euclidean distances
- Linear, polynomial, multiple, stepwise, weighted regression
- Extended regression diagnostics
- Multivariate general linear model

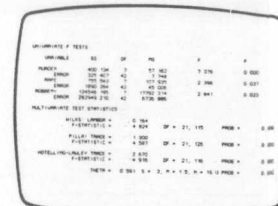
- Multi-way ANOVA, ANOCOVA, MANOVA, repeated measures, unbalanced designs, post-hoc tests
- Principal components with rotations and scores
- Multidimensional scaling
- Multiple and canonical discriminant analysis, Bayesian classification
- Canonical correlation
- Cluster analysis (hierarchical, single, average, complete median, centroid linkage, k-means, cases, variables)
- Nonparametric statistics (sign, Runs, Wilcoxon, Kruskal-Wallis, Friedman two-way ANOVA, Mann-Whitney U, Kolmogorov-Smirnov, Lilliefors, Kendall coefficient of concordance)
- Time Series (smoothing, seasonal and nonseasonal ARIMA, ACF, PACF, Cross-correlation function, transformations, forecasting, Fourier analysis)
- Nonlinear estimation (non-linear regression, least absolute values regression, logit, probit, maximum likelihood estimation, iteratively reweighted least squares)
- Supplements include: *Logit* (multinomial logit), *Probit* (probit analysis), *Testat* (classical and Rasch model test analysis), *Report* (formatter for SYSTAT files), *Transfer* (dBase, Lotus, GAUSS, STATA, SPSS direct file transfer)

SYSTAT operates on the following machines: IBM-PC/XT/AT, Apple II, Apple Macintosh, Kaypro, HP 150, HP 9000, DEC Rainbow, VAX, Alpha Micro, MS-DOS, C/P/M and UNIX.

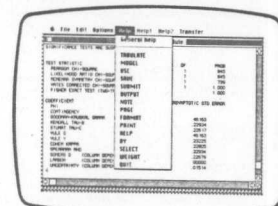
Single copy price:
\$595 USA and Canada / \$695 Foreign
Site licenses and quantity prices available
Call or write for additional information
SYSTAT, Inc.
2902 Central Street
Evanston, IL 60201
312 864.5670

SYSTAT

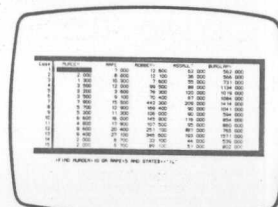
The system for statistics



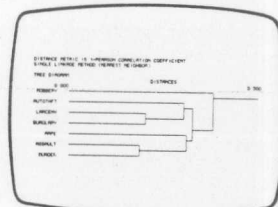
C/P/M



Apple Macintosh



VAX



IBM PC

Read more about the extraordinary success of SYSTAT in *So You've Got a Great Idea*, by Steve Filler (Addison Wesley, 1986).